Functional programmierung

How *map*, *filter* and *flatMap safe the day*

Imperativ VS. Functional

GO!

Imperativ

Computer as machine

Program as a sequence of commands

```
var x = 41
x++
```

Functional

Computer as evaluator of expressions

Program as a single expression

```
val a = x(y(z(param1)), f(param2))
```

Imperativ

Loops and jumps as basic building blocks

```
for(x in xs) {
    break;
}
```

Changing of storage cells as memory

```
var m = 0
m = 1
m = 2
```

Functional

Expressions as basic building block

```
x * y + z
```

Assignment of names to expressions as memory

```
val a = x * y + z
val b = a * 5
```

Imperativ

Functional

DRAW!

What is FP?

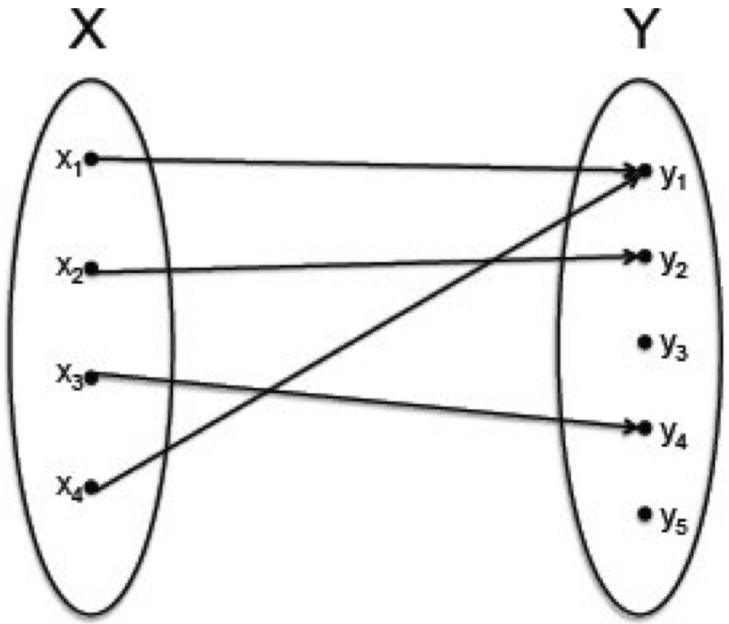
- Functions used as operations
- Functions used as abstractions
- Functions treated as values
- Functions returned as results
- Functions…

It’s all about functions!

What is a function?

From Wikipedia:

> A function f assigns to each element x of a source domain $\mathbb{D}$ exactly *one* element y of a target domain Z.



What is a function – in Kotlin?

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

```
fun sum(a: Int, b: Int) = a + b
```

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

Theory: referential transparency

- Describes a property of expressions:

```
val x = f(42)
val y = g(x)
val z = h(x)
```

=

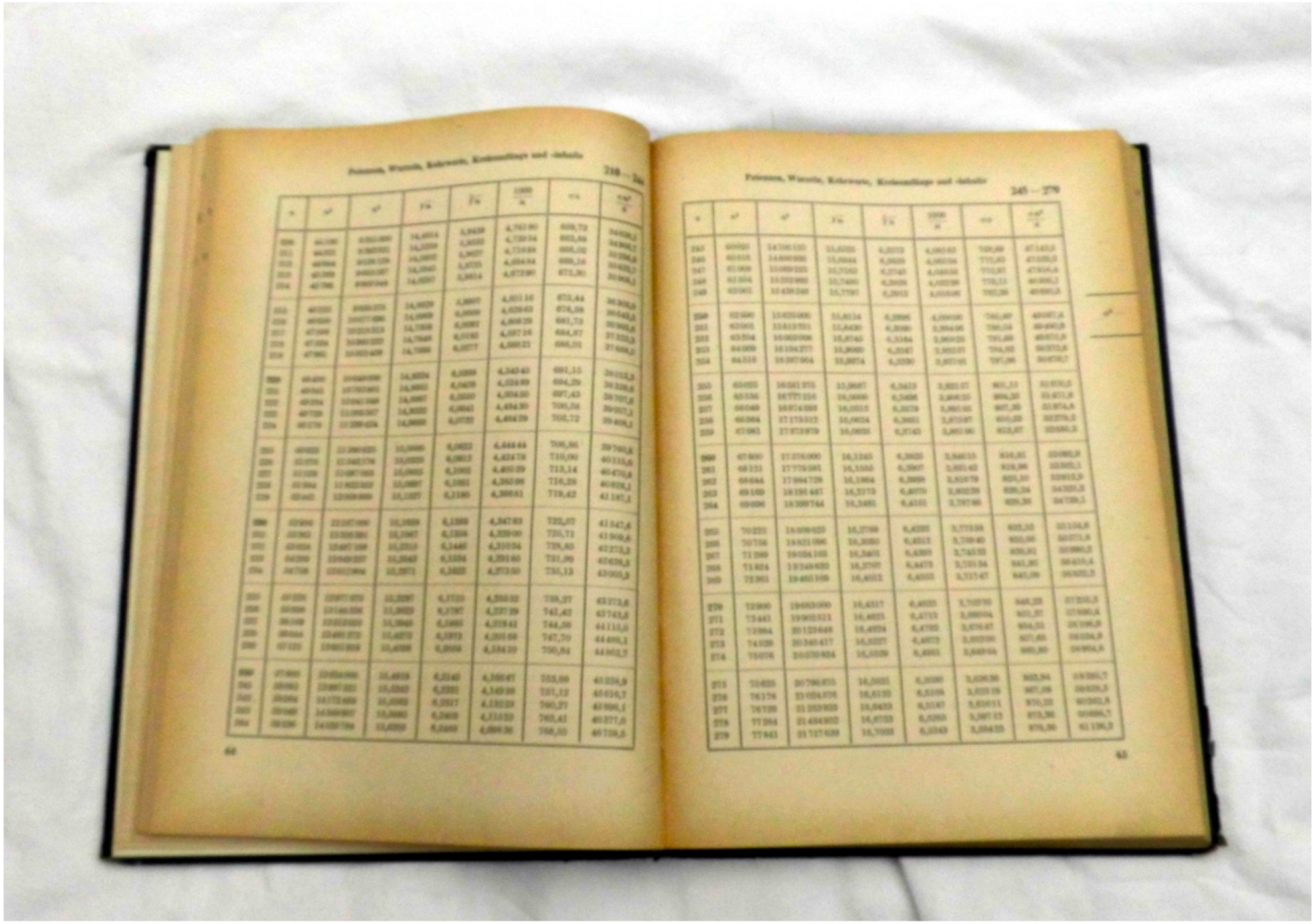
```
val y = g(f(42))
val z = h(f(42))
```

- (May) allow the proof of correctness
- Allows automated optimization like caching, lazy evaluation and parallelization

Functions as objects

How should one imagine such a thing?

Logarithms book – An old ‘log’-function



A table...

x	y
1	1
2	4
3	9
4	16
5	25

A schredder!

Working with functions

```
fun f(x: Int): Int = …
fun g(y: Int): String = …
fun h(z: String): Int = …
```

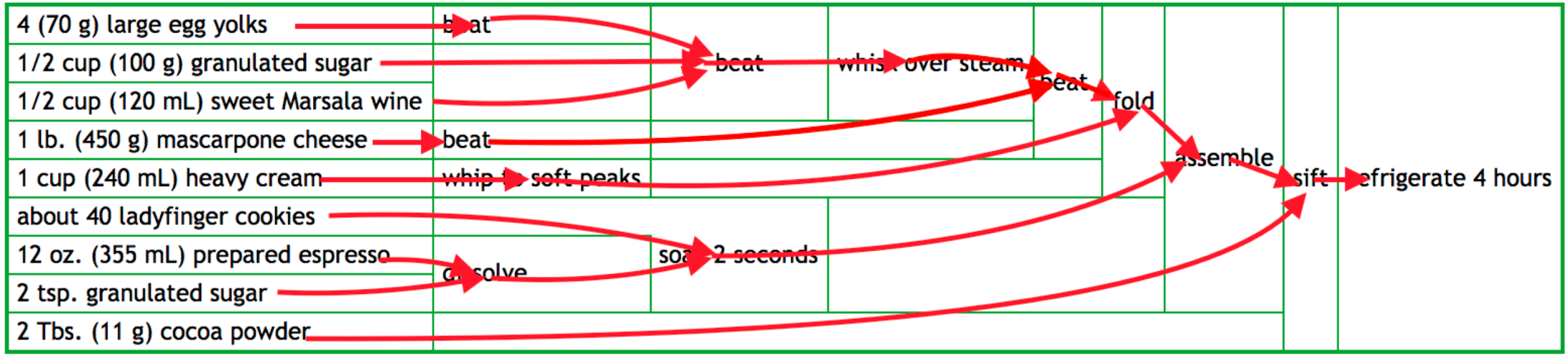
```
val x:Int = 10
val y:Int = f(x)
val z:String = g(y)
val a:Int = h(z)
```

```
val a:Int = h(g(f(10)))
```

Working with functions – making Tiramisu

4 (70 g) large egg yolks	beat	beat	whisk over steam	beat	fold	assemble	sift	refrigerate 4 hours
1/2 cup (100 g) granulated sugar								
1/2 cup (120 mL) sweet Marsala wine								
1 lb. (450 g) mascarpone cheese	beat							
1 cup (240 mL) heavy cream	whip to soft peaks							
about 40 ladyfinger cookies		soak 2 seconds						
12 oz. (355 mL) prepared espresso	dissolve							
2 tsp. granulated sugar								
2 Tbs. (11 g) cocoa powder								

Working with functions – making Tiramisu



Working with functions – mousse ou chocolat

```
fun schoggiMousse_machen(eier: Eier, zucker: Zucker, rahm: Rahm,
schokolade: Schokolade): SchoggiMousse {
    val (eiweiss, eigelb) = trennen(eier)
    val eischnee = schlagenBisFestUndGlänzend(
                        mischen(zucker, schlagen(eiweiss))
    )
    val grundmasse = schlagenBisLuftigUndHellgelb(mischen(zucker, eigelb))
    val schlagrahm = schlagenBisFest(Rahm)
    val schockoGrundmasse = mischen(grundmasse, schmelzen(schokolade))
    return kühlen(unterheben(
             eischnee, unterheben(schockoGrundmasse,schlagrahm))
    )
}
```

What is a lambda?

- A block with curly braces

```
val lambda: (Unit) -> Unit = { }
```

- Optionally with a parameter list

```
val param: (Int) -> Int = { p -> p }
val params: (Int, String) -> Int = { i, s -> i + s.length }
```

- Parameter list allows destructuring

```
val tuples: (Pair<Int, Long>, String) -> Long = { (i, l), s -> i + l +
s.length }
```

- Implicit first parameter ‘it’ when there is no parameter list

```
val withIt: (Int) -> String = { it.toString() }
```

What is a lambda?

- Result

```
val lambda = { 10 } // Last expression is the result

lambda() == 10
```

- Return?

```
fun funktion() : Int {
    listOf(100, 100).forEach {
        return 42
    }
    return 10
}
```

Anonymous functions

- Created with the ‘fun’ keyword

```
fun name() : Int {
    return 42
}
```

```
val anonym = fun() : Int {
    return 42
}
```

- Either a single expression or a block

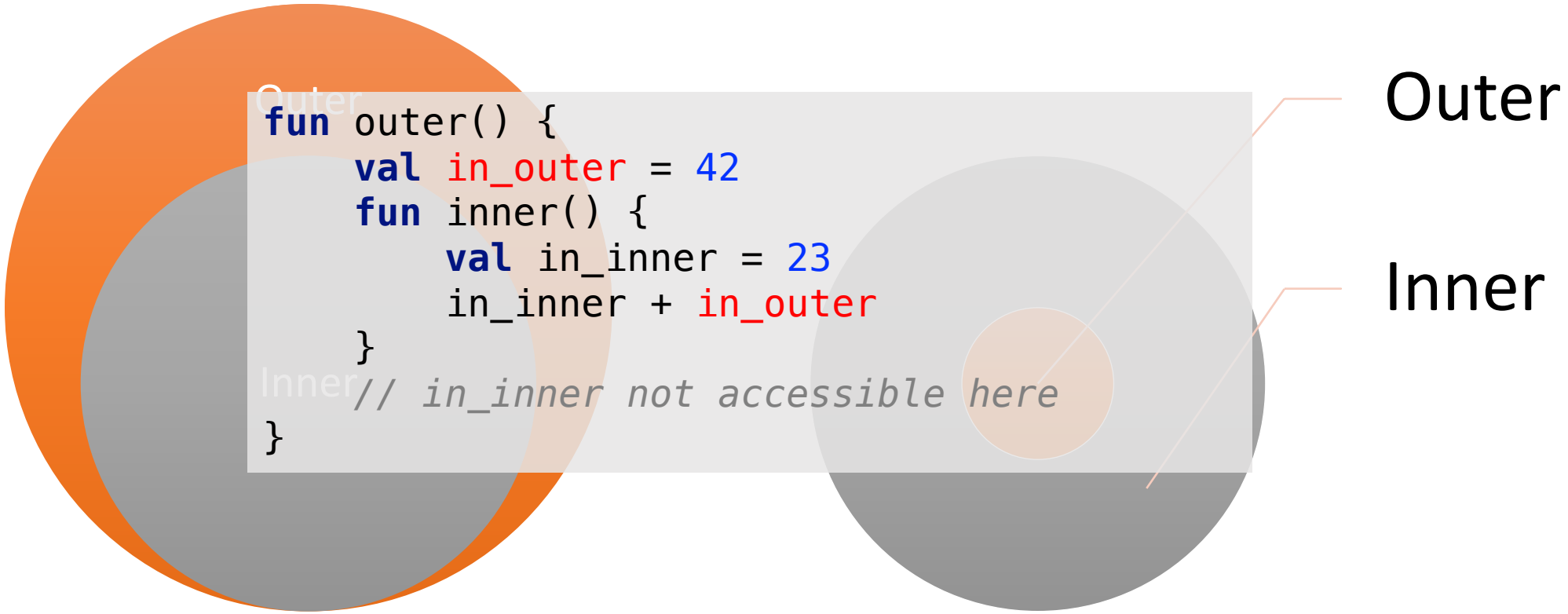
```
val anonym = fun() : Int {
    return 42
}
```

```
val anonym2 = fun() : Int = 42
```

Scope

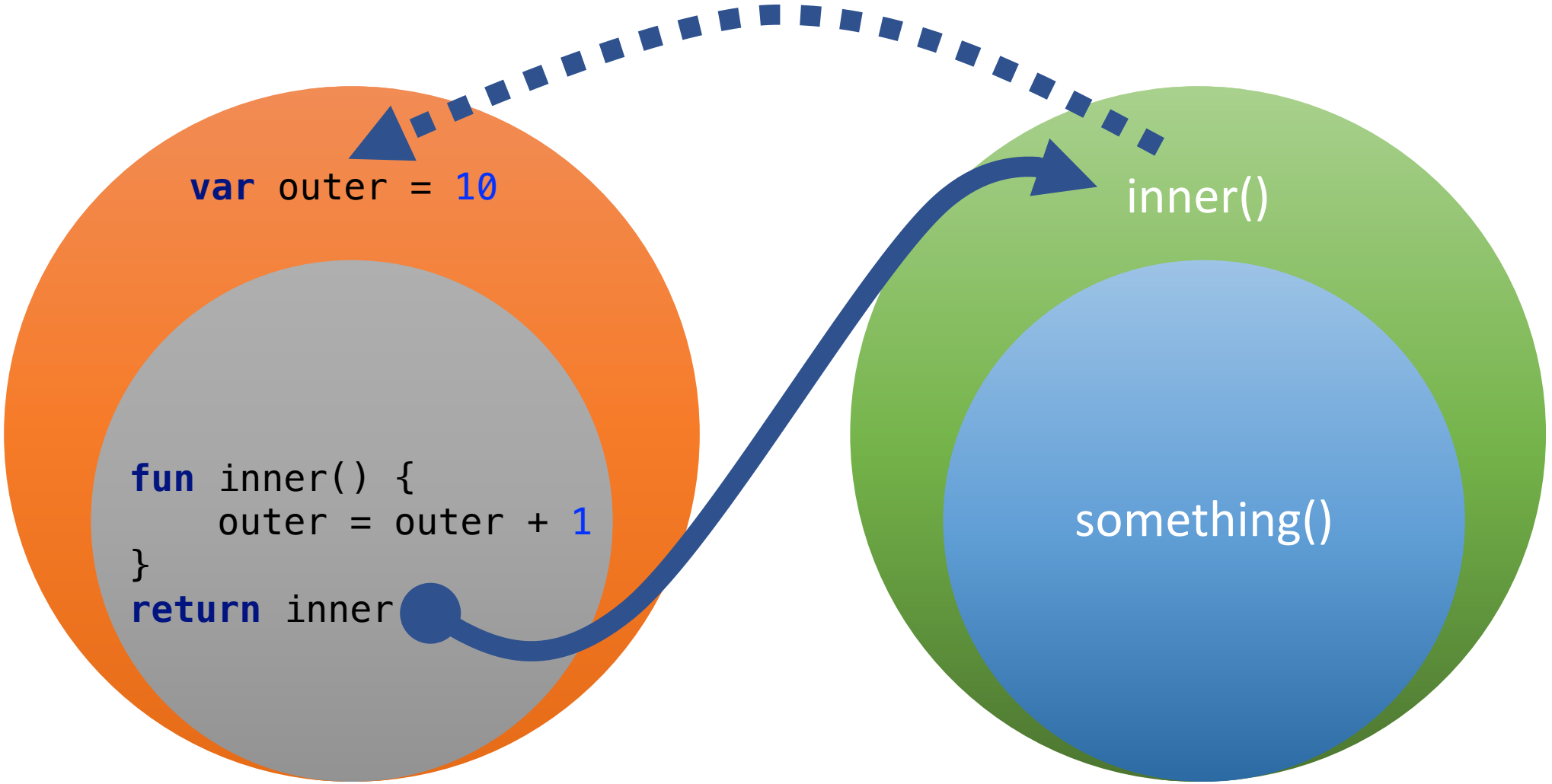
Function Nesting

Scope Nesting




```
fun outer() {
    val in_outer = 42
    fun inner() {
        val in_inner = 23
        in_inner + in_outer
    }
    // in_inner not accessible here
}
```

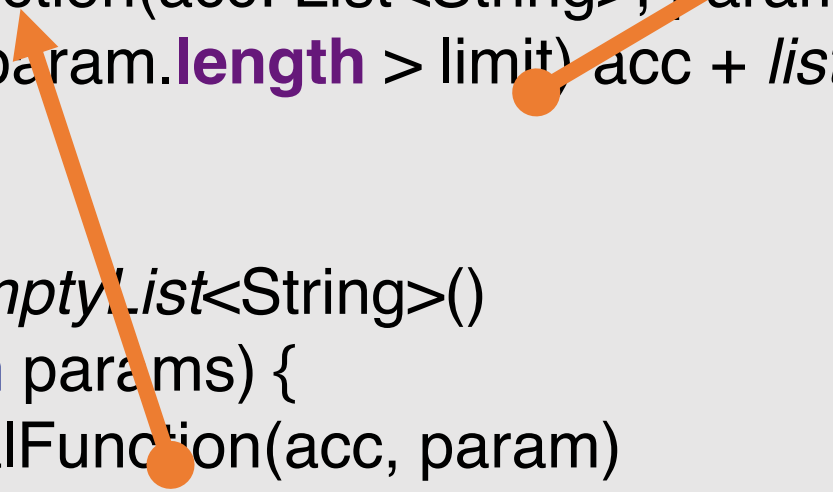
Closure


var outer = 10
fun inner() {
outer = outer + 1
}
return inner
inner()
something()

Local functions

```
fun someFunction(params: List<String>, limit: Int): List<String> {
    fun localFunction(acc: List<String>, param: String): List<String> {
        return if(param.length > limit) acc + listOf(param) else acc
    }

    var acc = emptyList<String>()
    for(param in params) {
        acc = localFunction(acc, param)
    }
    return acc
}
```



Functions as abstractions

Higher-Order functions allow the implementation of behavior on the base of other functions

→HOF are an instance of the strategy-pattern – each strategy is a function

Add abstractions - mousse ou chocolat

```
fun schoggiMousse_machen(eier: Eier, zucker: Zucker, rahm: Rahm,
schokolade: Schokolade): SchoggiMousse {
    val (eiweiss, eigelb) = trennen(eier)
    val eischnee = schlagenBisFestUndGlänzend(
                        mischen(zucker, schlagenBisFest(eiweiss))
    )
    val grundmasse = schlagenBisLuftigUndHellgelb(mischen(zucker, eigelb))
    val schlagrahm = schlagenBisFest(rahm)
    val schockoGrundmasse = mischen(grundmasse, schmelzen(schokolade))
    return kühlen(unterheben(
             eischnee, unterheben(schockoGrundmasse,schlagrahm))
    )
}
```

Making chocolate-mousse

```
<T> fun schlagen(
            was: T,
            bis: T -> Boolean = istFest) {
    …
}
```

With abstractions - mousse ou chocolat

```
fun schoggiMousse_machen(eier: Eier, zucker: Zucker, rahm: Rahm,
schokolade: Schokolade): SchoggiMousse {
  val (eiweiss, eigelb) = trennen(eier)
  val eischnee = schlagen(mischen(zucker,schlagen(eiweiss)),
                  bis = masse -> isFest(masse) && isGlänzend(masse)
    )
    val grundmasse = schlagen(mischen(zucker, eigelb),
                  bis = masse -> isLuftig(masse) && isHellgelb(masse)
    )
    val schlagrahm = schlagen(Rahm)
    val schockoGrundmasse = mischen(grundmasse, schmelzen(schockolade))
    return kühlen(unterheben(eischnee,
                    unterheben(schockoGrundmasse, schlagrahm))
    )
}
```

Higher order functions

Operations on Kotlin collections

- Map
- Filter
- FlatMap

Map – processing elements

```
fun <in A, out B> map(elements: List<A>, f: (A) -> B) : List<B> {
    var bs = listOf<B>()
    for(element in elements) {
        bs = listOf(f(element)) + bs
    }
    return bs
}
```

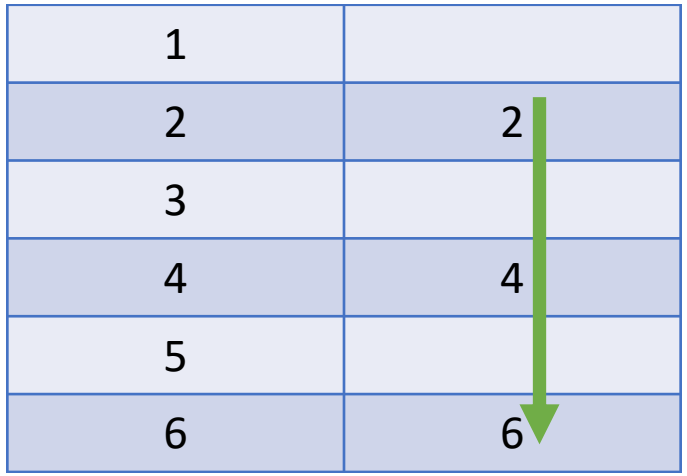
```
map(listOf(1, 2, 3, 4, 5, 6), { it * 10 }) == listOf(10, 20, 30, 40, 50, 60)
```

1	10
2	20
3	30
4	40
5	50
6	60

Filter – selecting elements

```
fun <A> filter(elements: List<A>, p: (A) -> Boolean): List<A> {
    var filtered = listOf<A>()
    for (element in elements) {
        if (p(element))
            filtered = listOf(element) + filtered
    }
    return filtered
}
```

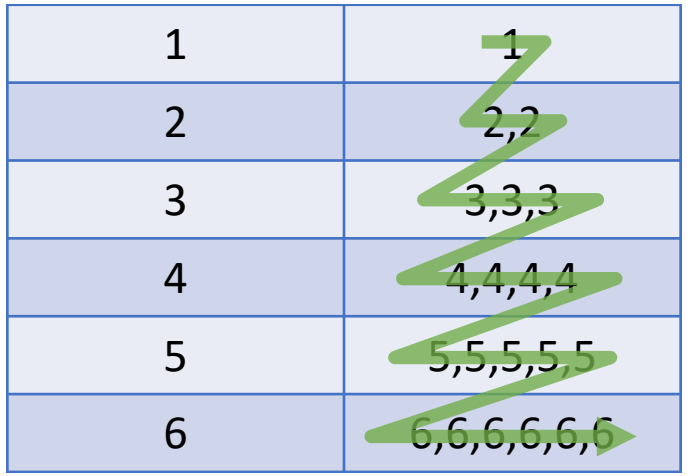
```
filter(listOf(1, 2, 3, 4, 5, 6), { it % 2 == 0 }) == listOf(2, 4, 6)
```



FlatMap – multiplying elements

```
fun <in A,out B> flatMap(elements: List<A>, f: (A) -> List<B>): List<B> {
    var bs = listOf<B>()
    for (element in elements) {
        bs = f(element) + bs
    }
    return bs
}
```

```
flatMap(listOf(1, 2, 3, 4, 5, 6), { fillWithElements(length = it, value = it} )
       == listOf(1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6)
```



Exercise

Implement Map and Filter using FlatMap

→ In the Kotlin REPL of AndroidStudio

Hint: Map & Filter with FlatMap

```
listOf(1,2,3).map { it * 10 }

listOf(1,2,3).flatMap { ??? }

listOf(1,2,3,4,5).filter { it % 2 == 0 }

listOf(1,2,3,4,5).flatMap { ??? }
```

Inlining

- Problem: Calling a function does not come for free
  - Prepare parameters: allocation, boxing, etc.
  - Acutal call
  - For Android: Dex-Limit
- Inline avoids this overhead
- Still not free, but a different currency:
  Code size → Compiler inserts the code at the call sites

Inlining: example

```
fun <T> lock(lock: Lock, body: () -> T): T {
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

```
lock(l) { foo() }
```

Inlining: manual variant

```
l.lock()
try {
    return foo()
}
finally {
    l.unlock()
}
```

Inlining: example

```
inline fun <T> lock(lock: Lock, body: () -> T): T {
    // ...
}
```

Inlining: limitations

- Does not work for local functions
- Does not work for functions containing local functions
- Inline automatically includes function paramtens – the keyword ‘noinline’ allows to disable this behavior

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
    // ...
}
```

Sources & informations about FP

Sources

- www.lihaoyi.com/post/WhatsFunctionalProgrammingAllAbout.html

Blogs & Videos

- https://medium.com/@JorgeCastilloPr/kotlin-functional-programming-does-it-make-sense-36ad07e6bacf
- https://medium.com/@BladeCoder/exploring-kotlins-hidden-costs-part-1-fbb9935d9b62
- https://www.youtube.com/user/DrBartosz/playlists → Category Theory